International Journal of Software Engineering and Its Applications
Vol. 7, No. 3, May, 2013

Visualizing and Analyzing the Structure of AspectJ Software under the Eclipse Platform

Sassi Bentrad [1] and Djamel Meslati [2]

Computer Science Department, LRI Laboratory
Badji Mokhtar-Annaba University, Annaba, Algeria
[1] sassi_bentrad@hotmail.fr, [2] meslati_djamel@yahoo.com

Abstract

Software is naturally intangible and abstract which makes the understanding task difficult. There is a growing need for visualizations that improve the comprehensiveness of its structure, behavior and evolution. Graphically visualizing abstract concepts provides a way to raise the abstraction level and therefore, to reduce the software complexity. The graphical visualization has an important contribution by presenting the software under an abstract synthetic view that gives a quick idea of its content, logic, structure and its entities' relationships. It is widely accepted that it can represent a valuable support during the development and maintenance processes. As AspectJ is a relatively new language with powerful specific constructs, it deserves support tools to visualize its software systems. This paper presents our recent work in software visualization with respect to analyze and visualize the AspectJ software structures using graphical elements well-known from daily life such as the Polymetric View and the City Metaphor to conduct various powerful analyses and permit an intuitive understanding of a given visualization and therefore, to get quickly an overview of a huge and complex software. VizzAspectJ-2D and VizzAspectJ-3D are two tools support we have built on top of the Eclipse platform respectively for the 2D and 3D visualizations.

Keywords: *Software maintenance, Program comprehension, Program analysis, Software visualization, Software metrics, Visualization metaphor, AspectJ*

1. Introduction

Today industrial systems require more complex software development with high qualities in terms of maintainability and reusability. The maintenance cost of a software system is frequently estimated between 50% and 75% of its total cost [1], and more than a half of the maintenance effort is devoted to understanding the application itself. Consequently, the understanding phase becomes a crucial issue whose enhancement highly motivates scientific work in the context of visualization [2].

Software visualization depends on paradigms. It must evolve as paradigms used to develop software systems evolve. In this context, both object-oriented development and aspect-oriented development have become standard for software development. The concepts introduced by these approaches have provided an answer to many problems of software engineering. In particular, they allow a more accurate modeling of reality, a better control of software complexity, and an easier reuse of artifacts produced. Although these approaches are successful in the production of complex software, practical experience with large projects has shown that programmers still face difficulties in maintaining their code [3].

Therefore, new methods and new assessment and understanding tools of software are needed to help both managers and developers in the maintenance activities. It helps to better understand and perceive the relationships (*i.e.*, dependencies) and interactions within the

software. In this domain, the information space visualized is highly structured, but a complexity arises from the fact that there is no single structure for this space but there are several significant or relevant structures from the user point of view. Since the data space is often extremely rich, understanding the complexity of the observed phenomenon depends on how the wealth of this space is returned.

Designing a software deals with the creation of abstractions which are inherently intangible and abstract and, therefore, difficult to apprehend and understand. They often involve a complex human communication in order to describe and convey their underlying meanings. This complexity increases according to the number of software entities (*i.e.*, objects or artifacts) and their relationships. Moreover, software artifacts are not static. As the software evolves, artifacts like documentation are no longer synchronized with the code. Consequently, they become useless and many maintenance tasks are directly performed using only the software source code. Since it is difficult to get an overall idea and reach a certain level of abstraction about the structure of software by reading its source code, the graphical visualization becomes a mandatory alternative. Abstract concepts of the graphical visualization provide synthetic forms with a suitable level of abstraction that are able to reduce the complexity of software systems [2].

1.1. Motivation and Objectives

The Aspect-Oriented Programming (AOP) approaches are relatively a new paradigm and the birth of a new paradigm requires new tools that allow on the one hand, to ensure its integration in the industry and, on the other hand, to provide the necessary means for software engineering developers and also researchers to study and understand it.

In this work we are interested in the AspectJ language as an important implementation of the AOP paradigm. This choice is motivated by the fact that it is a relatively young technology with powerful concepts. Even if the first operational AOP tools were not introduced until the late 90's, the adoption of the AOP approach is quite fast and its dissemination in the software engineering community is high. AspectJ can be integrated at low cost by companies expanding their tools [4]. For example, the AOP for Java is integrated into Eclipse via a plug-in that allows use of the AspectJ language.

Almost all software visualization tools are stand-alone applications. This forces the user to switch between them and the software editors. An Eclipse plug-in allows the use of visualizations without switching between the editor and another tool. So for this reason we have chosen to build support tools of the proposed approach on top of the Eclipse IDE in the form of plug-ins.

The main purpose in developing these tools is to further improve the ability to get an overview of software written in AspectJ and also to allow a quick understanding of its structure through a variety of visualization views.

1.2. Contributions

Firstly, we mention that this article is an extended version of our previous original work published as a research paper in [5].

This article presents the result of a project dedicated to the visualization in 2D and 3D of AspectJ software. While the 2D visualization provides a useful viewpoint when software are not complex, the visualization using the 3 dimensions leads to better utilization of space, since we can represent many concepts in exploiting the third dimension, and provides the opportunity to have better interactions. Our approach is based on a city metaphor as in the

case of the CodeCity tool of Richard Wettel [6] (*i.e.*, visualization of software as if it was a city).

Our 3D visualization is based on a city metaphor with a combination of some visualization techniques and software metrics. Our approach has been implemented as Eclipse plug-ins that are integrated seamlessly within the Eclipse Workbench. This is to be contrasted with most tools designed for the software visualization which are developed as a standalone application that force the user to switch between different windows and contexts. This change of context is time consuming and less friendly. By integrating our plug-ins within the Eclipse platform we make considerable steps to bring the visualization tools in the development process where users can view and analyze the representation of the source code while writing it.

The contribution of our work can be summarized in the creating of two tools that work in the Eclipse platform as a set of plug-ins, VizzAspectJ-2D and VizzAspectJ-3D respectively for the 2D and 3D visualizations. These support tools allowed us to get an overview of a complex AspectJ software with a quick and intuitive understanding of its structure in various 2D and 3D views. It efficiently integrates some features of sophisticated interaction modes to help the user interact with an effective and simple manner within the visualization views.

For the visualization in 3D, the real contribution is centered on the metaphor that we have used and the types of interaction modes we have implemented to assist the maintenance engineers to understand a large program written in AspectJ and also to interact with it during the programming process.

1.3. Overview of Paper Contents

The remainder of the paper is structured as follows. In Section 2 and as a background, we introduce the visualization domain and particularly software visualization and then we cite briefly some previous related works. In Section 3, we describe in detail our approach for visualizing AspectJ software, we illustrate the visual metaphors used, metrics selected to get an effective visualization and its explanation in 2D and 3D visualizations views. In Section 4, we describe our current work in more detail. For each support tool we have developed, we expose its architecture, design and its implementation. In order to experiment better the functionalities of these tools and validate them, section 5 presents a preliminary assessment we have conducted through case studies. Section 6 discusses briefly the presented work and then summarizes the possible future extension and finally in Section 7 we conclude the paper.

2. Background

2.1. Software Visualization

The important role that visualization plays in human reasoning in general and in scientific progress in particular has been emphasized by philosophers throughout the centuries. As a consequence visualization has become a discipline of computer science. Gershon [7] defines visualization as follows: "Visualization is more than a method of computing. Visualization is the process of transforming information into a visual form, enabling users to observe the information. The resulting visual display enables the scientist or engineer to perceive visually features which are hidden in the data but nevertheless are needed for data exploration and analysis."

Researchers in programming languages have tried to design a structured programming language to be easier to use and understand, but they were limited by the structure of the text alphabet. The software visualization as a sub domain of information visualization has the

same objective of facilitating the use and improving the comprehensibility, but it does not suffer the same limitations since several graphical constructs can be used.

The software visualization is seen as a process of abstraction of concepts: classes, packages, and interactions to a concrete and comprehensive picture. This picture shows some properties of the software. In general, visualization is needed to express and simplify an abstract concept or a real object in a way that the user of this visualization can easily understand. Particularly in object-oriented software systems, this kind of visualization is useful for modeling the properties of a specific element (*i.e.*, the contents of a class) or properties of a specific system (*i.e.*, dependencies between classes, hierarchies, packages, *etc.*).

The ability to quickly browse a huge software system and detect defects highlighted by visualization or having the opportunity to understand its form to quickly discover its structure and dependencies is a main feature in terms of comprehensibility and maintenance. But how the source code and its semantic meaning can be transformed into a graphical representation?

Many authors define software visualization as the visualization of algorithms and programs (a narrow definition). By this definition, the field of software visualization can be divided into two separate zones [8, 9]:

- **Program Visualization.** Refers to the visualization of source code or data structures in a static or dynamic form.
- **Algorithm Visualization.** Refers to the visualization of high-level abstractions that describe the software. A good example of this is the dynamic visualization of an algorithm and its implementation. It allows visualizing the general behavior of an algorithm using a data abstraction, but also the operations performed on them.

However, a better and wider definition considers that the software visualization is the visualization of artifacts related to software and its development process (a wide definition). In addition to the program code, these artifacts include requirements and design documentation, changes to the source code, and bug reports, for example. In fact, researchers in software visualization develop and investigate methods and uses of computer graphical representations of various aspects of software, for example its static structure, its concrete and abstract execution, and its evolution. In a nutshell, they are concerned with visualizing the structure, behavior, and evolution of software.

- **Structure.** Refers to the static parts and relations of the system, *i.e.*, those which can be computed or inferred without running the program. This includes the program code and data structures, the static call graph, and the organization of the program into modules.
- **Behavior.** Refers to the execution of the program with real and abstract data. The execution can be observed as a sequence of program states, where a program state contains both the current code and the data of the program. Depending on the programming language, the execution can be viewed in a higher level of abstraction as functions calling other functions, or objects communicating with other objects.
- **Evolution.** Refers to the development process of the software system and, in particular, emphasizes the fact that program code is changed over time to extend the functionality of the system or simply to remove bugs. To this effect and for this aspect, the visualization sometimes includes an animation that shows how software evolved in several steps and thus retraces the history of the development or maintenance.

2.2. Related Works

In general, software visualization efforts have a long history. A lot of different support tools of visualizations have been developed so far, but just a handful of them are not academic experiments without any practical application. In this sub-section we provide briefly background information on some previous related works. Among the tools that were the source of inspiration for our work: for the 2D Visualization were Softwarenaut [10] and CodeCrawler [11]. For the 3D Visualization were the tool CodeCity [12], and the tool Sv3D [13, 14].

2.3. AspectJ, an Aspect-Oriented Programming Language

AOP is becoming an increasingly popular programming methodology; we can find implementations of AOP for many modern languages. For the Java language, AspectJ is a general-purpose, aspect-oriented implementation that has the largest community acceptance [15]. AspectJ is a popular choice for several good reasons. One of its strengths is, and always has been, its pragmatic approach to language design. Instead of allowing the language to get bogged down in theory, AspectJ's developers started with basic AOP support and added new features only after people in the field had discussed their practical use extensively.

The result was the creation of a simple language that was powerful enough to solve real problems. Another real strength of AspectJ is the tool support that is so crucial to every developer. Let's face it—not many of us write code that runs perfectly the first time, and debugging is an activity on which we spend a good portion of our working life. AspectJ is integrated with several IDEs such as Eclipse, NetBeans, JBuilder, and Emacs JDEE [15].

AOP and AspectJ's influence on software development has just begun. It is going to have an impact on virtually every kind of programming: enterprise applications, desktop clients, real-time systems, and embedded systems [4]. AOP is still new and as with any new paradigm, it will take time to be assimilated into the programming community.

3. Proposed Approach to Visualize AspectJ Software

3.1. Software Metrics

A metric is formally defined as an association between the empirical world and the digital world. It is a number or a symbol associated with an entity to characterize its attributes. According to Fenton [16], we seek to formalize and to better understand the world around us through a series of metrics. These metrics allow us to validate our intuitions from this world. What we must accurately represent the observed entities and the relations which stand them to each other.

Metrics are an important tool in summarizing large amounts of information and are extremely useful because they express digitally something that is not necessarily a number. It increases and summarizes a particular aspect of an entity, by providing an opportunity to have a significant representation of that aspect in an entire graphical representation.

Software metrics are a special kind of analysis focused on the structure of the source code. Classic software metrics range in variety from the very simple Source Lines of Code (SLOC) to more complex measures such as Cyclomatic Complexity measurements. Such metrics are widely used to judge the quality of source code.

3.2. Visual Metaphors: Polymetric View and City Metaphor

Software Visualization is not done for analysis purpose only, or to reflect each detailed concept in software. Instead of all that, its essential aim is to increase the software

understanding. In this sense, metaphor-based visualization is an effective way helping in the representation and analysis of large object-and aspect-oriented software systems. It uses resources from human visual and cognitive systems to extract from the visualization views regularities and discontinuities that are the basic elements of any qualitative study [17, 18].

- **Polymetric View.** A visualization that exploits different metrics in order to show a set of software entities is said to be polymetric. In order to understand the architecture of programs, Lanza [19] uses polymetric views enriched with software metrics extracted from the source code of the program to be visualized. These views represent a static display where software's entities such as Classes and Aspects are modeled as rectangles, using some metrics as shown in Figure 1.

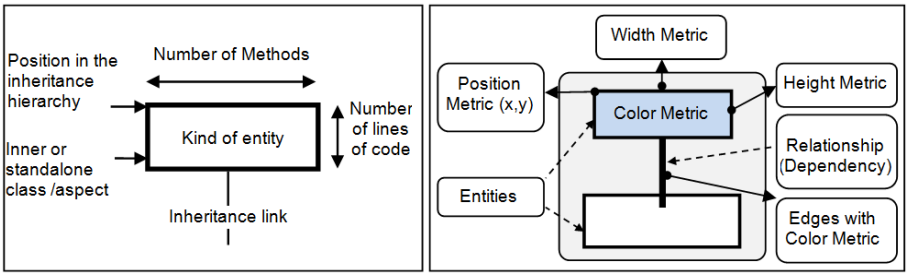


Figure 1. A Simple Polymetric Visualization / Common Metrics Semantic in a Polymetric View

- **City Metaphor.** A visual metaphor is an analogy which underlies a graphical representation of an abstract entity or concept with the goal of transferring properties from the domain of the graphical representation to that of the abstract entity or concept. Metaphors can make an interface more intuitive and more attractive. However they can also add an unnecessary visual noise [20]. The attractive aspect of this kind of visualization is interesting to attract users and developers in particular.

  The preferred metaphor and the well-known from daily life is that of the city. Although several studies have been done using this metaphor, some are superficial by simply having a correspondence between the graphical elements. Most are not oriented for software quality, but rather for the representation of software entities. Work would be focused on the semantic similarities between the use of districts and buildings in a city and use of packages and classes in software [21]. A screenshot of a city metaphor is presented in Figure 2 [6].

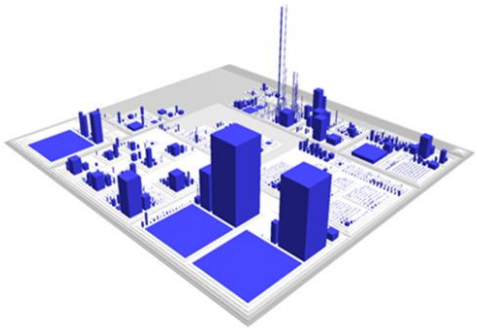


Figure 2. A Software System Visualization in CodeCity

In this section, we present our visualization approach. Notice that in our work, an important effort was dedicated to the selection of software metrics that allow an effective visualization. We present them first, and then we describe the proposed approach and give a preliminary assessment.

3.3. Important Parameters for an Effective Visualization

After a careful study of a set of AspectJ programs we selected metrics that we consider highly discriminating and suitable for a helpful visualization. Table 1 summarizes some of these metrics. Notice that these metrics corresponds to the main concepts of the Java language and the concepts introduced by AspectJ. We have coded these metrics using visualization metrics that preserves scalability and efficiency of the visualization tools while being user friendly. The aspect metrics are defined and its computation is based on the reflection mechanism of the AspectJ language itself.

3.4. Metrics Expressed in Visualization: Metrics Explanation

The plug-in uses different metrics in its views, modeling Classes, Aspects and Packages entities as nodes (rectangles) according to the view in which they are represented. It models metrics selected in Table 1, Class and Aspect Type, Hierarchy, Dependency and their weights.

Table 1. Metrics Selected for Visualization

Acronyms	Descriptions
N_MDc	Number of methods declared in a class **c**
N_ADc	Number of attributes declared in a class **c**
N_MDa	Number of methods declared in an aspect **a**
N_ADa	Number of attributes declared in an aspect **a**
N_ITMDc	Number of methods introduced in a class **c** *(inter-type declaration)*
N_ITADc	Number of attributes introduced in a class **c** *(inter-type declaration)*
N_POINTCUTa	Number of pointcuts declared in an aspect **a**
N_ADVICEa	Number of advices declared in an aspect **a**
. . .	. . .

3.4.1. 2D Visualization: For a 2D graphical representation of these metrics, we use a hierarchical visualization technique to model entities (such as classes, interfaces, aspects and packages, *etc*.) using an abstract form like rectangles. Dependencies between entities (such as hierarchy) are modeled using edges. Although this choice may seem straightforward, it's really an excellent compromise between simplicity and quantity of information that can be expressed. The other metrics are represented using the position, the color, rectangles height and width and so on.

Several tools dedicated to the visualization of Java code already exist. We make use of the X-Ray plug-in [22], which provides various views with adaptable parameters. In the following, we first describe how X-Ray represents each Java software metric using Position, Color, *etc.*, then we show how this approach has been adapted to deal with the Aspect code.

Table 2 summarizes the metrics applied to the nodes of a Java code in the "System Complexity" view.

Table 2. Node Metrics in the "System Complexity" View for Java Code

Graphical Attributes	Descriptions (Corresponding Java Software Metrics)
Position	Computed according to the disposition of the inheritance tree (top-bottom oriented).
Color	*Class Type:* Green (external class to the project), White (interfaces), Blue (concrete) and Light blue (abstract).
Width	Number of attributes and methods.
Height	Number of lines of code.
Border	It can be black for a stand-alone (autonomous) class or orange for an inner class.
Edges	Inheritance between classes (the dependencies of the inheritance hierarchy).

The metrics applied to the nodes of an Aspect code in the "System Complexity" view are given in Table 3.

Table 3. Node Metrics in the "System Complexity" View for Aspect Code

Graphical Attributes	Descriptions (Corresponding AspectJ Software Metrics)
Position	Computed according to the disposition of the inheritance tree (top-bottom oriented).
Color	*Aspect type:* Orange (external aspect to the project), Pink (concrete) and Light pink (abstract).
Width	Number of attributes and methods.
Height	Number of pointcuts and advices.
Border	It can be black for a stand-alone (autonomous) aspect or blue for an inner aspect.
Edges	Inheritance between classes and aspects (the dependencies of the inheritance hierarchy: aspect-aspect/aspect-classes dependency).

Figure 3 reflects graphically the metrics used in the "System Complexity" view. Five entities are represented (4 classes and an interface), and three edges. Notice that the external node has default width and height.

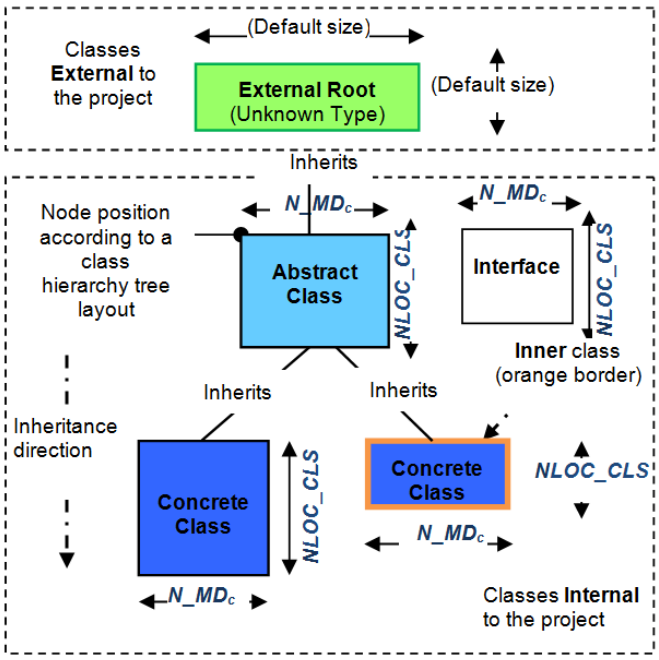


Figure 3. Nodes Metrics of the "System Complexity" View (Java Code)

Figure 4 reflects graphically the metrics used in "System Complexity" view. There are 4 entities represented (4 aspects), and three edges. Notice that the external node has default width and height.

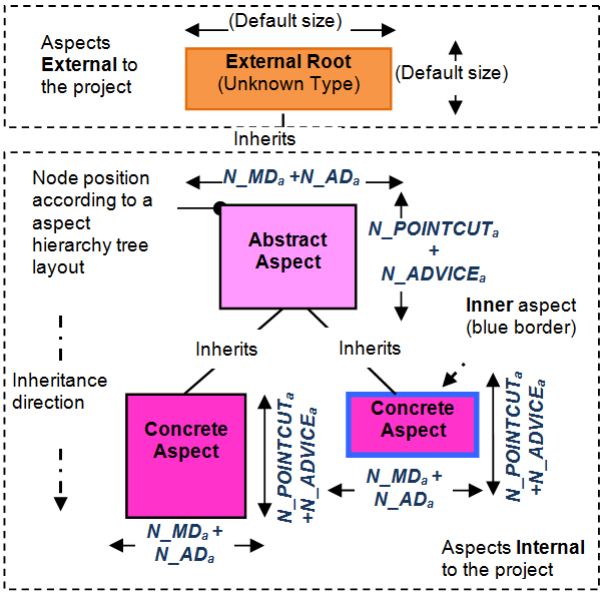


Figure 4. Nodes Metrics of the "System Complexity" View (Aspect Code)

Table 4 summarizes the metrics applied to nodes in "Classes, Aspects & Packages Dependencies" views.

Table 4. Node Metrics in the "Classes, Aspects & Packages Dependencies" Views

Graphical Attributes	Descriptions (Corresponding Java and AspectJ Software Metrics)
Color	*Package:* brown; *Class:* white (interface), light blue (abstract), and blue (concrete); *Aspect:* light pink (abstract), and pink (concrete).
Edges	Dependencies between entities (colors and thickness described in the Table 5).

Figure 5 graphically depicts the metrics used in the "Classes, Aspects & Packages Dependencies" views; the view (A), the view (B), represented three entities (classes) and three entities (aspects), respectively. All of them have the same size and their color reflects their class or aspect type. There are three arrows between them, highlighting dependencies of different strength, according to Table 5.

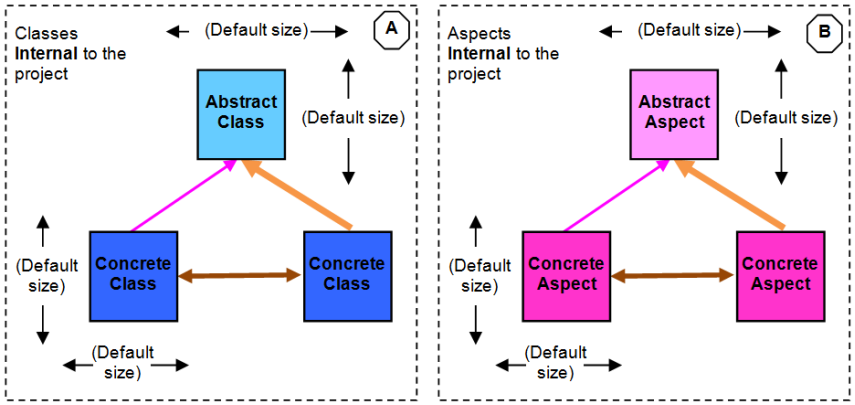


Figure 5. Nodes Metrics of the "Classes, Aspects & Packages Dependencies" Views

There are six levels of dependencies, as shown in Table 5.

Table 5. Edges (arrows) Metrics in "Classes, Aspects & Packages Dependencies" Views

Level	Nbr. of External Calls	Color	Width
1	**1-2**	Light pink	Default, thin size (2 pixels)
2	**3-4**	Pink	Augmented by a factor of 2 (4 pixels)
3	**5-9**	Orange	Augmented by a factor of 3 (6 pixels)
4	**10-19**	Light Brown	Augmented by a factor of 4 (8 pixels)
5	**20-49**	Dark Brown	Augmented by a factor of 5 (10 pixels)
6	**≥ 50**	Black	Augmented by a factor of 6 (12 pixels)

3.4.2. 3D Visualization: For a 3D graphical representation, we use a city metaphor that is similar to the one used by CodeCity tool where entities (classes, interfaces and aspects) are represented as "buildings", and packages as "districts" in the city.

In AspectJ, the implementation of the weaving rules by the compiler is called Static Crosscutting (SC) that affects the static structure—the classes, interfaces, and aspects—of the program and this is done using forms called Introduction mechanism; the weaving rules cut across multiple modules in a systematic way in order to modularize the crosscutting concerns [15].

In this visualization, we used different metrics and we distinguish between two phases: before and after the SC in the program in two 3D views, see Figure 6.

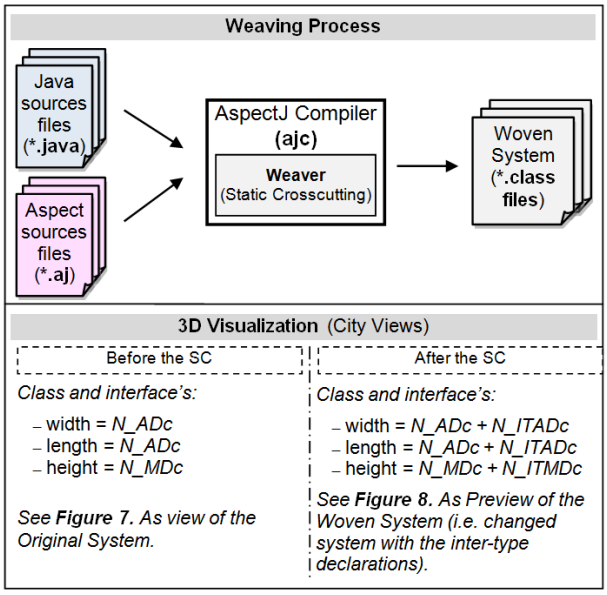


Figure 6. Explanation of the 3D Visualization Before and After the Weaving Step

- **Explanation of the 3D view before the SC.** (See Figure 7)
  - *Classes* and *interfaces* are represented as *buildings* where the *width* and *length* correspond to *N_ADc* metric and the *height* correspond to *N_MDc* metric.
  - *Aspects* are represented as *buildings* where the *width* corresponds to *N_ADa*, the *length* corresponds to *N_MDa* and the *height* corresponds to *N_POINTCUTa* + *N_ADVICEa*.
  - *Packages* are represented as *districts*.
  - *Colors* for the *Java class type*: blue (concrete), light blue (abstract), white (interface) and green (class external).
  - *Colors* for the *Aspect type*: pink (concrete), light pink (abstract) and orange (aspect external).

- **Explanation of the 3D view after the SC.** (See Figure 8)

After the SC, the change will be in the part of Java code. Classes and interfaces are represented as buildings.

In AspectJ, the weaving process is carried out during the compilation phase. The AspectJ compiler (ajc) processes the source and byte code for the core concerns and the programmatic expression of the weaving rules in the aspects. It then applies the rules to all the modules and creates output class files.

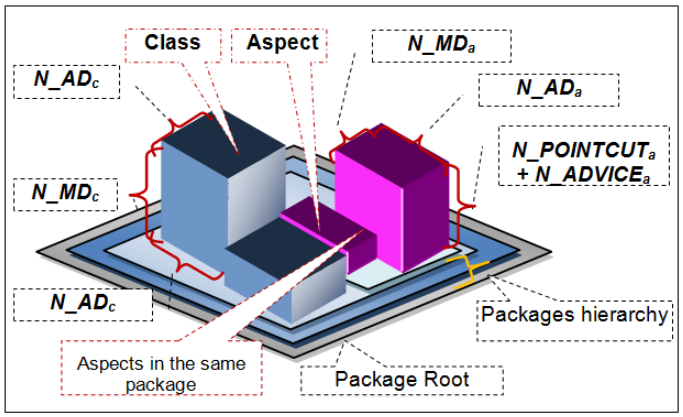


Figure 7. Nodes Metrics of the City View Before the Static Crosscutting

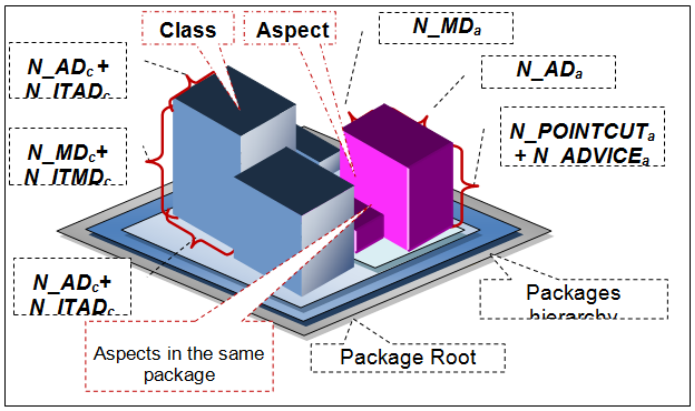


Figure 8. Nodes Metrics of the City View After the Static Crosscutting

4. Current Work: Architecture, Design and Implementation

In this section, we describe briefly our support tools that have been developed. For each one, we expose its architecture, design and its implementation.

4.1. Architecture

The Figure 9 below illustrates the general architecture of our 2D and 3D Visualization Tools under the Eclipse Platform: VizzAspectJ-2D and VizzAspectJ-3D. These tools are evolving as a collection of Eclipse plug-ins. Let us now look deeper into it.

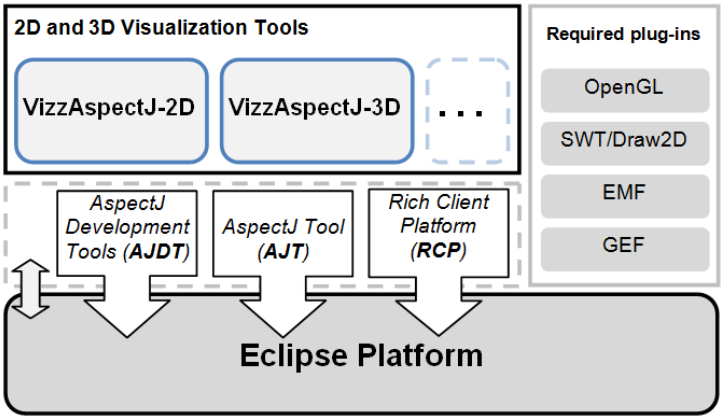


Figure 9. The High-Level Architecture of the Visualization Tools under Eclipse Platform

In the following sub-sections we will expose these tools at a somewhat technical level.

4.2. Design and Implementation

The continual release of new tools, either prototypes or commercial products, precludes any study to get enough distance on the field of visualization. Software visualization tools allow the programmer to raise the level of abstraction. Almost all these tools are standalone applications; this forces the user to switch between them and their favorite code editor. The development of an open-source software of visualization as a plug-ins integrated in the Eclipse platform is an important step towards the visualization tools in the development process. So that the user can analyze and visualize the software while he is writing it.

We have built on top of the Eclipse platform VizzAspectJ-2D and VizzAspectJ-3D respectively to visualize the AspectJ software structures in 2 and 3 dimensions. In the following, we present these tools at a somewhat technical level.

4.2.1. The 2D Visualization Tool -- VizzAspectJ-2D

- **Production Chain**

Under VizzAspectJ-2D, various 2D visualization views are built according to a production chain as shown in Figure 10.

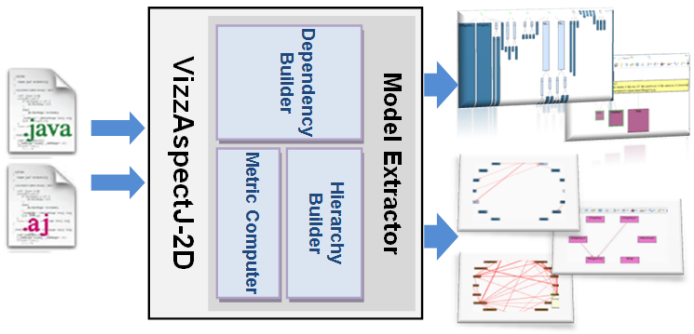


Figure 10. Production Chain of a 2D Views under VizzAspectJ-2D

- **Internal Code Representation (ICR)**

One of the most important part of the tool is the Model Extractor, responsible for creating an Internal Code Representation (ICR) which reflects the underlying source code. It is composed of 2 phases: the first is divided into two subtasks, the Hierarchy Builder and Metric Computer, while the second is the Dependency Builder.

- **Model Extractor.** This is the first task performed by the tool immediately after being initialized. It is responsible for analyzing the source code of the project that the user wants to visualize. The class that performs this analysis is ModelExtractor, its subtasks (i.e. the Hierarchy Builder and Metric Computer) and the Dependency Builder have been created as separated threads, in order to minimize the impact of their intensive tasks on the overall Eclipse user interface.
- **Hierarchy Builder.** It parses the project and collects information about the inheritance hierarchy of every class and aspect of the project. The ICR is a set of data structures containing meaningful data about the project, packages, classes and aspects; moreover it stores every metric and dependency. Once it performed its task, the project, previously seen as a set of files and directories, slowly takes shape.
- **Metric Computer.** It is responsible for collecting information about the metrics used by the "System Complexity" view. It retrieves the number of methods, attributes, number of lines of code, etc... As soon as the Metric Computer thread finishes its task, the Model Extractor has filled the ICR with all the information needed by the "System Complexity" view to create and show its graphical representation (See Figure 11).
- **Dependency Builder.** It scans the source code of the project and collects information about dependencies between classes and aspects. These dependencies will be used by "Packages, Classes and Aspects Dependencies" views while creating dependency edges (arrows) between entities (See Figure 12).

- **VizzAspectJ-2D Model Creation**

VizzAspectJ-2D creates a model of the code starting from the source files. The source files are analyzed and important values are gathered from it. The model created in which classes and aspects are organized in a hierarchy based on inheritance. After the tree is built, metrics computed (stored in the ICR) are applied to its representation. At this point, the tool could be able to visualize the "System Complexity" view with limited functionalities without the dependency-related actions (See Figure 11).

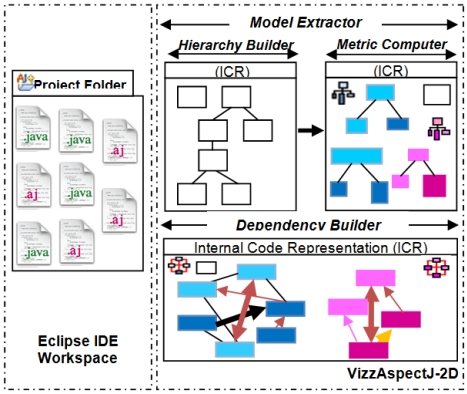


Figure 11. Creation of the Internal Code Representation under VizzAspectJ-2D

As soon as the Dependency Builder collected all the data about the metrics used by the "Class, Aspect and Package Dependencies" Views, the ICR contains all the information needed to create and visualize all its views. The Figure 12 shows how this is done.

To improve the usability of VizzAspectJ-2D, we create an Incremental Dependency Builder that will execute on a limited set of entities on request; we had the possibility to execute it on request.

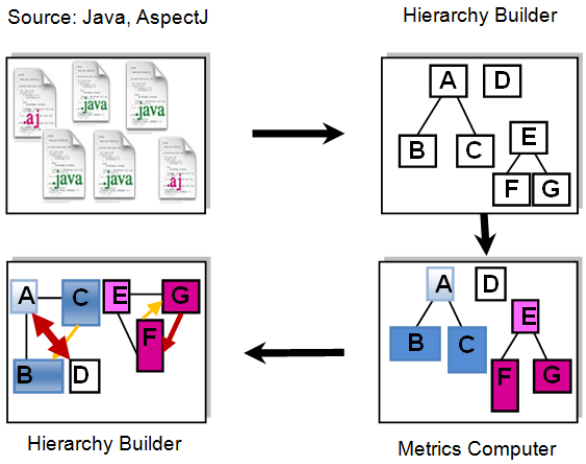


Figure 12. Construction Cycle of a 2D Views under VizzAspectJ-2D

- **UML Class Diagram**

The most important part of the tool is its core, the Internal Code Representation (ICR). To better understand its components, let us include a UML class diagram modeling the classes that are involved. Figure 13 gives the main idea how the VizzAspectJ-2D Core is structured.

This diagram shows how every AspectJ entity is modeled by an EntityRepresentation. Furthermore, packages, classes and aspects are contained within another entity (respectively, a project and a package), therefore we created the ContainedEntityRepresentation class. The ModelExtractor class contains a ProjectRepresentation that is made up by zero or more PackageRepresentations that are composed of zero or more ClassRepresentation and zero or more AspectRepresentation.

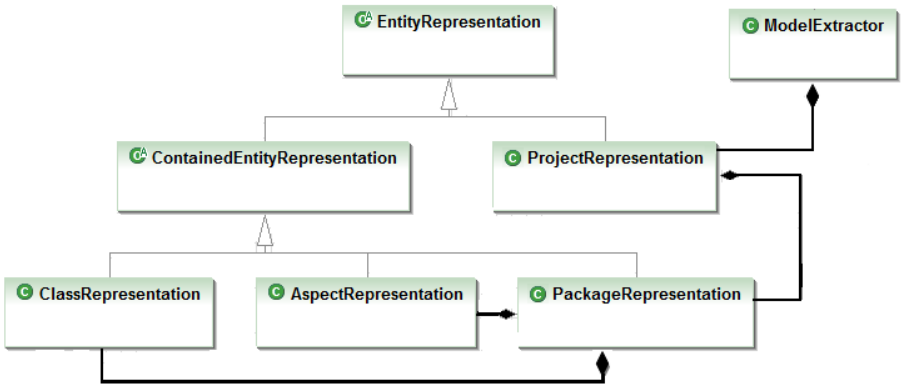


Figure 13. A View of the UML Class Diagram for the Classes Involved in the ICR

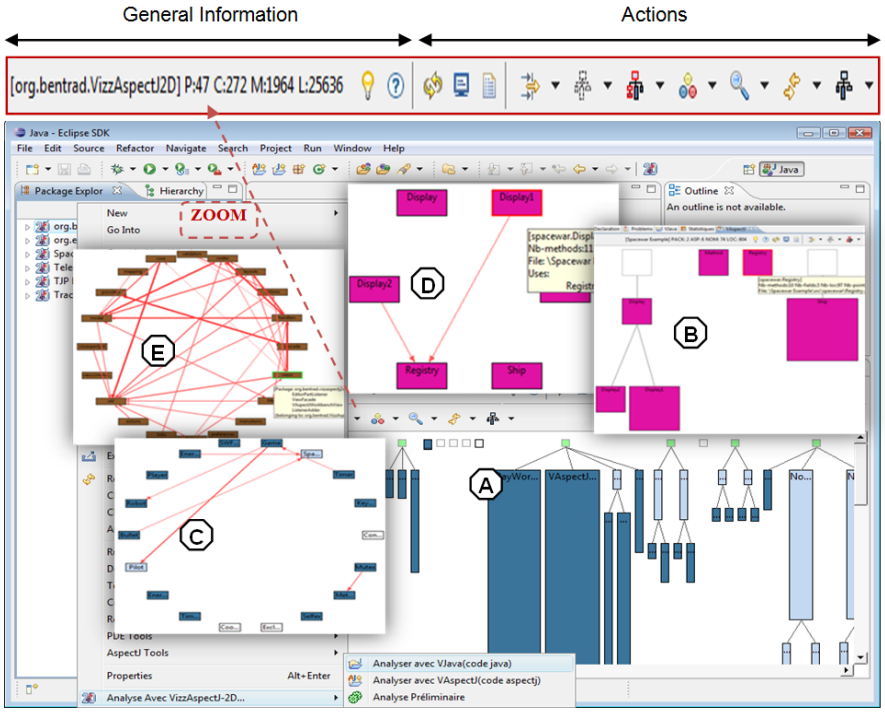
- **The User Interface**

The user interface of VizzAspectJ-2D is shown in Figure 14. Interaction takes place in the editor pane which is situated on the top side of the view. It provides general information about the analyzed project and a collection of actions to the user. Some actions are available for each visualization view, while others are specific to certain views.

This VizzAspectJ-2D plug-in provides 2D polymetric views such as:

– **The "System Complexity" view.** It is particularly efficient while spotting disharmonies in the design and implementation of a system. It is easy to find and identify big nodes (compared to the others) or anomalies in the shape of the project (provided by the inheritance tree). The user is therefore able, with a single picture, to analyze and understand complex systems in terms of some metrics without the need of reading source code.

– **The "Classes, Aspects and Packages Dependencies" views.** They are arranged in a bi-dimensional circle where the entities (packages, classes or aspects) are linked together by dependency links, each of them with a certain weight, highlighting how strength is the dependency between entities. Nodes and edges are displayed following the metrics explained previously.



Legend:
A – System Complexity View for Java code.
B – System Complexity View for Aspect code.
C – Classes Dependencies View.
D – Aspects Dependencies View.
E – Packages Dependencies View.

Figure 14. The User Interface of VizzAspectJ-2D and the 2D Visualization Views

For the Java code part of an AspectJ program, we consider that a class has a dependency when it uses some code implemented by another class. For the Aspect code part, we consider that an aspect has a dependency with a class (resp. another aspect) when it contains a join

point concerning another class (resp. another aspect) or introduces an attribute or method to another class (resp. another aspect).

This visualization highlights anomalies design, providing information about coupling and cohesion. The dependencies views of classes, aspects and packages highlight the effect that an entity may have on another when they work together to achieve a specific functionality.

- **SWT/Draw2D Implementations**
  - **Draw2d** is a layout and rendering toolkit building on top of SWT inside the Eclipse platform in combination with the GEF.
  - **Graphical Editing Framework Draw2d.** GEF is interactive Model-View-Controller (MVC) framework, which fosters the implementation of SWT-based tree and Draw2d-based graphical editors for the Eclipse Workbench UI. For more information see [26].
  - **SWT.** Also known as the Standard Widget Tool, a general, platform independent, UI library providing graphical 2D widgets (Lists, Figures, Labels, *etc.*). All the graphics created by the tool has been implemented using SWT components.

4.2.2. The 3D Visualization Tool -- VizzAspectJ-3D

- **Production Chain**

Under VizzAspectJ-3D, various 3D visualization views are built according to a production chain as shown in Figure 15.

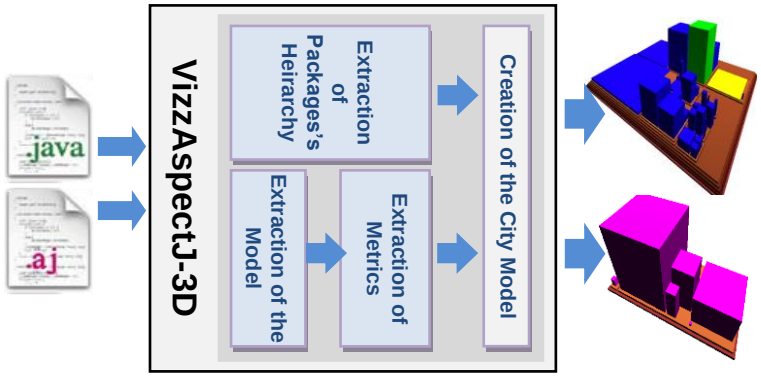


Figure 15. Production Chain of a 3D Views under VizzAspectJ-3D

- **UML Class Diagram**

From the Figure 16 we can see a simplified UML Class diagram of the classes that models the city in order to ease the understanding of how we have modeled the city.

This diagram gives the main idea how the VizzAspectJ-3D Core is structured. There's the main component called City that contains a tree of set of CityEntity. The tree represents the hierarchical structure of packages and sub-packages. The Layouts inheriting from Layout contain the behavior of how the entities in the city should be placed.

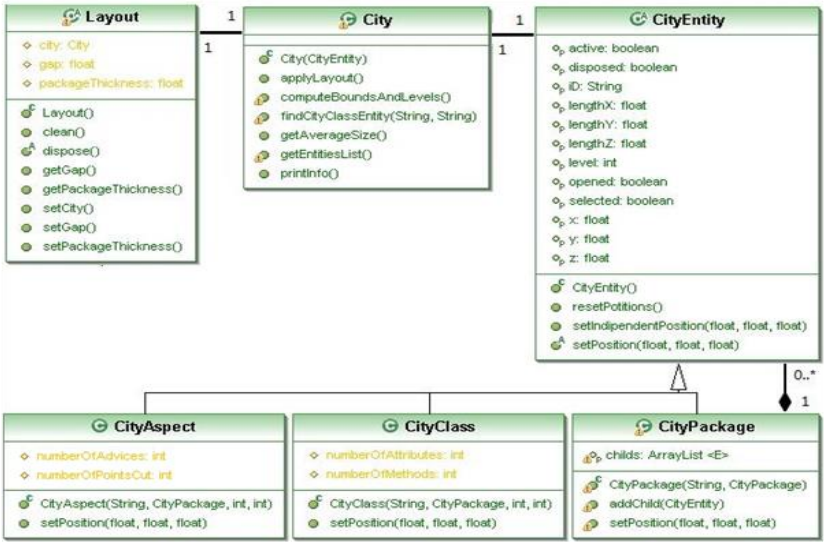


Figure 16. A View of the UML Class Diagram of the Model of the City

From the choice of the graphical implementation many other classes that have high importance have to exist (*i.e.*, the Drawer to draw the entire city on the Eclipse view and the Camera for navigability). Other classes can be created for handling the interaction between plug-ins, models, framework, *etc.*.

- **City Creation**

VizzAspectJ-3D interacts with the Eclipse IDE, plug-ins of the tool VizzAspectJ-2D and the OpenGL library in order to display the analyzed project. The Figure 17 explain in details, at a high level, the interactions between all the plug-in.

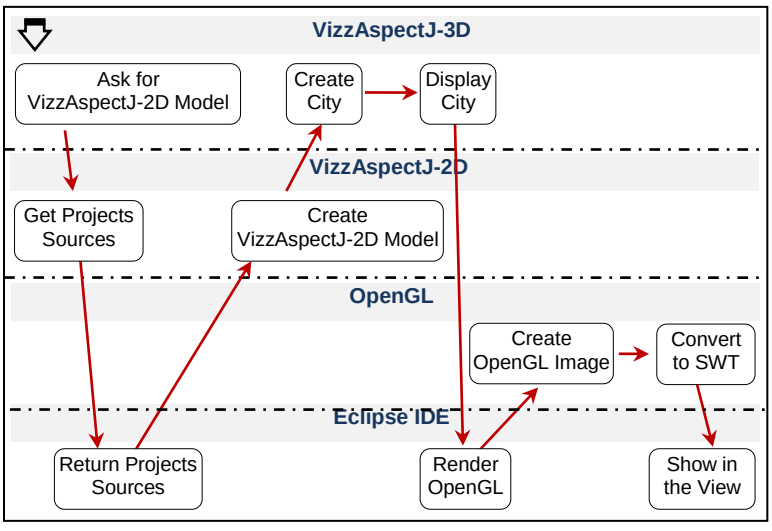


Figure 17. Process of Creating a 3D City View

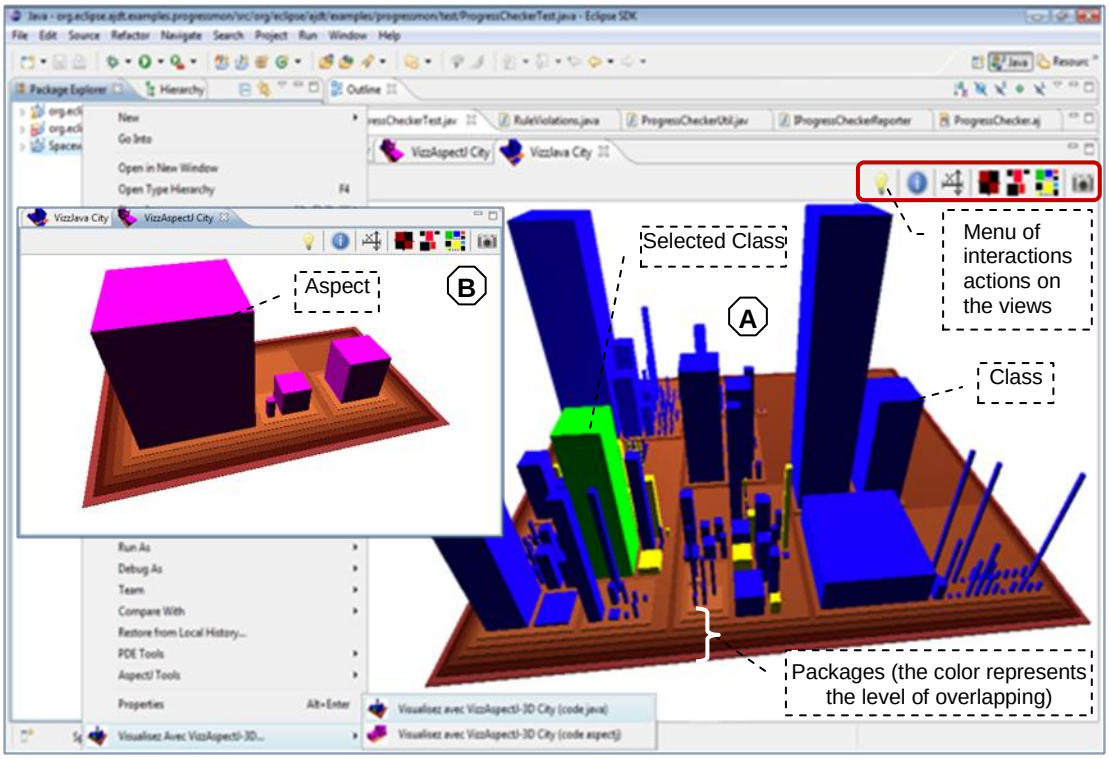
The tool VizzAspectJ-3D needs some functionalities of VizzAspectJ-2D in order to create its model (*i.e.*, the VizzAspectJ-2D model). When this model is obtained, the city representation of the classes and aspects is built.

First, from the VizzAspectJ-2D model we can get the list of all the packages, since they are not hierarchically organized we have to structure them according to an algorithm. From this hierarchical model of packages, districts and sub-districts hierarchy was created, just by mapping each level in the hierarchy to a district.

Under VizzAspectJ-2D, each representation of a package contains a list of classes and aspects. From each class and aspect, we take the metrics we need in order to build the city, and create buildings from them.

- **The User Interface**

The tool VizzAspectJ-3D provides two Eclipse views of 3D visualization (using a city metaphor): the "VizzAspectJ City" view for the Aspect code part and the "VizzJava City" view for the Java code part of the program that the user wants to visualize (See Figure 18).



Legend:
A – VizzJava City View for Java code.
B – VizzAspectJ City View for Aspect code.

Figure 18. The User Interface of VizzAspectJ-3D and the 3D Visualization Views

- **OpenGL Implementations**

The most solution we have found working properly was an experimental deprecated OpenGL binding for SWT in the Eclipse IDE [27].

5. Assessment and Validating

We conducted an assessment of our tools presented above in order to experiment better functionalities and validate them through case studies consisting in well known examples of AspectJ programs. This allowed as getting an idea about the complexity of these programs and putting to the test our tools with real software examples.

5.1. Case Study 1 -- 2D Visualization Tool

As a first case study, we attempted to evaluate the tool VizzAspectJ-2D on the well known AspectJ programs. As shown in Figure 14, the 2D visualization tool is a stable prototype with limited functionalities but mature enough to be useful while analyzing small and medium sized AspectJ projects. With the view "System Complexity", we can detect identity disharmonies and identify design issues just by looking at the form of the visualized project. With the views "Classes, Aspects and Packages Dependencies" we can detect the collaboration disharmonies and identify incoming and outgoing dependencies.

Exploiting the abstractions provided by all the 2D Polymetric Views, we can analyze any AspectJ project at Class, Aspect and Package level, and also deepening our knowledge by browsing the dependencies and hierarchies.

5.2. Case Study 2 -- 3D Visualization Tool

In addition, we attempt to assess the tool VizzAspectJ-3D on the well-known AspectJ programs such as "AJHotDraw" [23]. The Table 6 below shows its main characteristics [24]. As shown in Figure 18, this program is viewed as a city, where Aspects are represented as pink buildings, Classes as blue buildings and Packages as districts.

Table 6. Table Shows the Main Characteristics of the AJHotDraw Program

Java Code Part				Aspect Code Part				
Nb. classes	Nb. interfaces	Nb. fields	Nb. methods	Nb. aspects	Nb. pointcuts	Nb. advices	Nb. introduced fields	Nb. introduced methods
255	49	708	3348	10	3	5	1	20

6. Discussion

Our software visualization was designed in a way that does not produce a high cognitive load, by using visualization presentation techniques such as color, size, shape, position and visual metaphor (*i.e.*, city metaphor). The user can interact with the generated views to get more information about the target software and also to get quickly an overview of it.

6.1. Interaction & Navigation

Being able to navigate and interact with a visual representation is a critical feature, since static pictures are limited with respect to expressiveness. We support the following types of interactions:

- **Selection.** In our tools one can select any artifact or group of artifacts and interact with them. The selection can be done manually by clicking on elements, or automatically with a query engine. With the current selection one can perform operations, such as adding to or removing from the selection, clearing the selection, and inverting the selection.
- **Spawning.** Spawning complementary views, *i.e.*, isolating elements is useful whenever we need to focus on a particular part of the system. We can make a selection of artifacts in the city and spawn a new view that contains only the selected elements, allowing us to continue the exploration in detail.
- **Tagging.** During the exploration of product views, we may need to tag a set of entities (buildings and rectangular nodes) because we want to remember them or because we deem them as less relevant. We can assign a particular color to a selection or also use transparency (this latter is part of our future work).
- **Filtering.** Manual selection of elements can be cumbersome, therefore we provide a means to perform automated searches by indicating a set of criteria, which enable searching for artifacts that match a particular name, type, category, or artifacts related to the current selection, *etc*. We implemented a query engine to automatically search for the elements matching the query.
- **Navigation.** The proposed tools provide various keyboard-and mouse-based navigations possibilities: moving back or forward, hovering left or right, orbiting around the city, changing altitude.

After using these tools for a certain time, we noticed a few aspects we want to discuss:

- **Scalability.** Because we settled our initial level of granularity to the class level, our approach scales up well in terms of the size of the system that we can display. However, in cities representing very large software systems the interactivity and navigability can be substantially slowed down. Performance optimization is mandatory to increase realism. We are currently considering the use of level-of-detail (LOD) techniques to improve scalability.
- **Completeness.** The classes, aspects and the package structure provide an overview of the system. At the current stage we do not directly represent lower-level artifacts, such as methods, attributes, pointcuts and advices that would actually reside within the buildings. We also do not currently directly represent relationships such as inheritance, method invocations and how classes are connected to aspects. While we have this information at disposition, and we can already represent them as edges connecting the buildings, they quickly lead to over-plotting problems. An appropriate representation of the relationships and the lower-level artifacts is part of our future work.

Through the studied examples, we noticed that the proposed tools were reliable and scale up well for relatively heavy real open source software. We have also tested these tools within

a development process for small AspectJ programs and we conclude that the visualization was helpful and can be used for the AspectJ language learning for beginners. We have yet considered some real applications, more metaphors are required for a better understanding for offering a simple and cognitive visual understanding of more huge software systems and the evaluation of the approach is still ongoing.

6.2. Future Work and Possible Extensions

Our tools introduced in this paper needs to be improved in the near future to adapt to users' needs, therefore we have in mind some future extension planned as follows:

- Improve the Graphical User Interface,
- Automatic Choice of Graphical Layout,
- Optimize the graphical representation of the metaphors used, and
- Visualizing the Behavior, and Evolution.
- For future extension we have planned to visualize the behavior and the evolution of Aspect-Oriented Software during the development process, in addition give the possibility to compare several versions of a software at the same time.
- Integrate these tools with other visualization plug-ins to provide to the user a high variety of visualization tools.
- Visualize other Languages, such as AspectC++, CaesarJ, JAC, *etc.*

  Our tools focus on AspectJ, but other Aspect-Oriented Programming languages could be visualized. There should be no difficulties in providing visualization for other available languages, and to visualize many different languages simultaneously.
- Provide the Visual Programming methodology.

  In order to provide Visual Programming for AspectJ. The graphical notations for AspectJ could also be manipulated, but that is beyond the scope of this work. Drag and drop techniques could rearrange source code at the statement level, and toolbars could automatically construct code for programming constructs or members of Classes and Aspects (*i.e.*, field, method, advice, pointcut, *etc.*) [25].

7. Conclusion

The software is naturally intangible and abstract. Moreover, its code is not a static artifact. It evolves and changes, implying that its design is revised and improved continuously, and it becomes increasingly complex and larger in size. Thus in practice, the development and maintenance processes are time-consuming because software complexity becomes more difficult to manage. Consequently designing and specifying the overall software structure emerges as a new kind of problem. The programmers who want to extend and maintain it must first understand it. That is where the visualization is useful.

However, visualizing software is not an easy task because of the huge amount of information comprised in the software. The visualization so far has concentrated mostly on the structure at various levels of abstraction. The static program visualization represents a way to visualize the structure of a program given; firstly based on the computation of static properties of the program and then visualize the results of static program analyses.

This work dealt with this kind of visualization. Our investigations in this field, and particularly that of the AspectJ software visualization, allowed us to make some contributions. We have presented a specific approach and discussed our recent work in software visualization with respect to visualize AspectJ software in 2 and 3 dimensions by means of a simple and well-known graphical elements recognized from daily life such as the Polymetric View and the City Metaphor respectively.

To support our proposed approach of visualization, we proposed two support tools that work inside the Eclipse platform as a set of plug-ins, VizzAspectJ-2D and VizzAspectJ-3D for analyzing and visualizing the structure of AspectJ software. These tools provide various 2D and 3D views with some features of sophisticated interaction modes that assist the maintenance engineers to interact with it during the programming process in a simple and effective manner, such as the zooming and filtering techniques.

The proposed tools allowed us to conduct different powerful analyses, further improve the ability to get quickly an overview of huge and complex AspectJ software and also to get an intuitive comprehensibility of its structure through the variety of visualizations views produced.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] I. Sommerville, "Software Engineering. Harlow", England: Addison-Wesley, 9. Edition, **(2010)**.
[2] R. Koschke, "Software visualization in software maintenance, reverse engineering, and reengineering: a research survey", Journal of Software Maintenance (JSM), vol. 15, no. 2, **(2003)**, pp.87-109.
[3] T. M. Pigoski, "Practical Software Maintenance: Best Practices for Managing your Software Investment", John Wiley & Sons, **(1997)**.
[4] R. Laddad, "AspectJ in Action: Enterprise AOP With Spring Applications", Manning Publications, 2nd edition, **(2009)**.
[5] S. Bentrad and D. Meslati, "2D and 3D Visualization of AspectJ Programs", In Proceedings of the 10th International Symposium on Programming and Systems (ISPS), Algiers, Algeria, **(2011)**, pp. 183-190.
[6] R. Wettel, M. Lanza, "Visualizing Software Systems as Cities", In VISSOFT. Edited by Maletic JI, Telea A, Marcus A, IEEE Computer Society, **(2007)**, pp.92-99.
[7] D. G. Nahum "From perception to visualization", In Scientific Visualization: Advances and Challenges, Academic Press, **(1994)**.
[8] J. Stasko, J. Domingue, M. H. Brown and B. A. Price (Eds), "Software Visualization: Programming as a Multimedia Experience", Cambridge, MA: MIT Press, **(1998)**.
[9] S. Diehl, "Software visualization: visualizing the structure, behaviour, and evolution of software", Springer, **(2007)**.
[10] Softwarenaut, http://scg.unibe.ch/softwarenaut/.
[11] CodeCrawler, http://www.moosetechnology.org/tools/retired/codecrawler/.
[12] CodeCity, http://www.inf.unisi.ch/phd/wettel/codecity.html.
[13] J. I. Maletic, A. Marcus and L.Feng, "Source Viewer 3D (sv3D) - A Framework for Software Visualization", In ICSE. Edited by Clarke LA, Dillon L, Tichy WF, IEEE Computer Society, **(2003)**, pp. 812-813.
[14] SV3D, [January 2012] [http://www.sdml.info/projects/sv3d/].
[15] R. Laddad, "AspectJ in Action: Practical Aspect-Oriented Programming", Greenwich, CT, USA: Manning Publications Co., **(2003)**.
[16] N. Fenton and S. L. Peeger, "Software Metrics: A Rigorous and Practical Approach", Boston, MA, USA: PWS Publishing Co., 2nd edition, **(2003)**.
[17] A. Marcus, L. Feng and J. I.Maletic, "Comprehension of Software Analysis Data Using 3D Visualization", In IWPC, IEEE Computer Society, **(2003)**, pp. 105-114.
[18] F. L. M. J. A. Marcus, "3D Representations for Software Visualization", Proceedings of the ACM Symposium on Software Visualization (SoftVis), San Diego, CA, **(2003)**, pp.27-36.

[19] M. Lanza and S. Ducasse , "Polymetric Views - A Lightweight Visual Approach to Reverse Engineering", IEEE Computer Society Trans. Software Engineering, vol. 29, no. 9, **(2003)**, pp. 782-795.
[20] G. Lako and M. Johnson, "Metaphors We Live By", Chicago, USA: University Of Chicago Press, 2nd edition, **(2003)**.
[21] G Langelier, H. A. Sahraoui and P. Poulin, "Visualisation et analyse de logiciels de grande taille", L'OBJET, vol. 11, no. 1-2, **(2005)**, pp. 159-173.
[22] XRay, http://atelier.inf.unisi.ch/~malnatij/xray.php.
[23] AJhotDraw, http://ajhotdraw.sourceforge.net/.
[24] J. Y. Guyomarch, "Une architecture pour l'évaluation qualitative de l'impact de la programmation orientée aspect", PhD thesis, Université de Montréal, Faculté des études superieures, **(2006)**.
[25] S. Bentrad and D. Meslati, "Visual Programming and Program Visualization: Towards an Ideal Visual Software Engineering System", International Journal on Information Technology (IJIT), vol. 1, no. 3, **(2011)**.
[26] Draw2D, http://www.eclipse.org/gef/draw2d/index.php.
[27] OpenGl, http://www.eclipse.org/swt/opengl/.

Authors

Sassi Bentrad is a Ph.D. student in Complex Software Engineering. He obtained his Master of Science degree in Computer Science from the University of Badji Mokhtar-Annaba (UBMA), Algeria in 2009. He completed his thesis under Prof. Meslati's guidance. His research interest include software visualization and visual programming for advanced separation of concerns.

Djamel Meslati is a professor in the department of computer science at the University of Badji Mokhtar-Annaba (UBMA), Algeria. He is the head of the research group on evolution and reuse of software systems at the Laboratory of Research on Computer Science (LRI). His research interests include software development and evolution methodologies and separation of concerns.